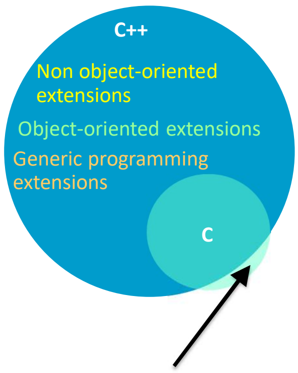
MODULE 1

C++ AS A BETTER C

C++ As a Better C



Minor exceptions:
C code that is not C++

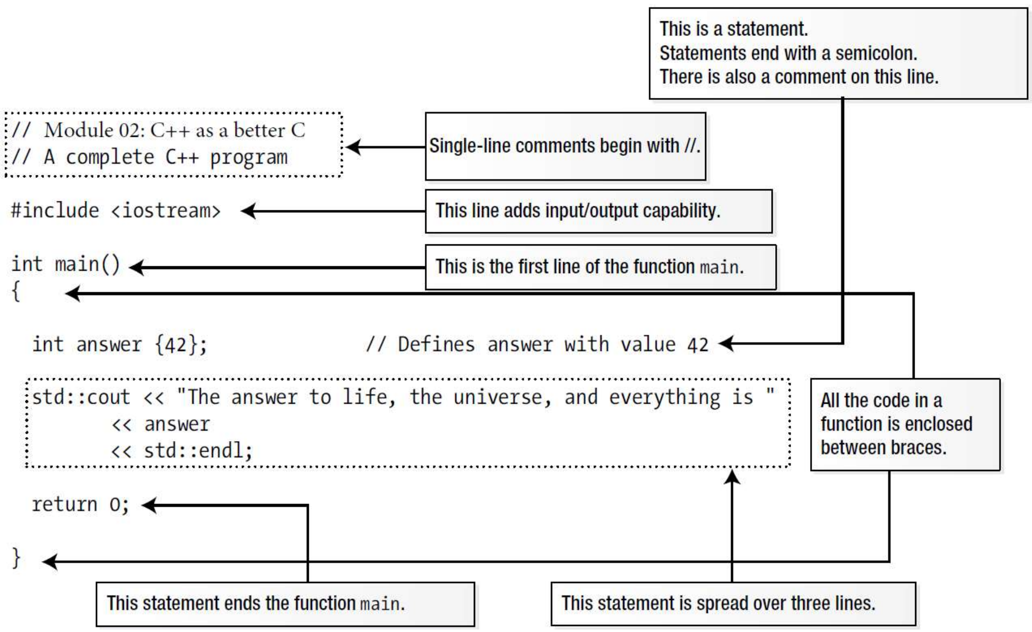
C++ As a Better C

> C++ was developed from the C programming language, by adding some features to it. These features can be collected in three groups:

1. Non-object-oriented features, which can be used in coding phase. These are not involved with the programming technique.
2. Features which support object-oriented programming.
3. Features which support generic programming.

> With minor exceptions, C++ is a superset of C.

C++ Program Concepts



C++'s Enhancements to C (Non-OO)

- **Caution**: The better one knows C, the harder it seems to be to avoid writing C++ in C style, thereby losing some of the potential benefits of C++.
  1. Always keep object-oriented and generic programming techniques in mind.
  2. Always use C++ style coding technique which has many advantages over C style.
- Non object-oriented features of a C++ compiler can be also used in writing procedural programs.

C++'s Enhancements to C (Non OO)

- Comment Lines
- /* This is a comment */
- // This is a comment
- C++ allows you to begin a comment with // and use the remainder of the line for comment text.
- This increases readability.

Declarations and Definitions in C++

> Remember; there is a difference between a declaration and a definition

> A declaration introduces a name – an identifier – to the compiler. It tells the compiler “This function or this variable exists somewhere, and here is what it should look like.”

> A definition, on the other hand, says: “Make this variable here” or “Make this function here.” It allocates storage for the name.

```
extern int i;       // Declaration
int i;                   // Definition
struct TComplex{    // Declaration
    float real,img;
};
TComplex c1,c2;     // Definition
void func( int, int);  // Declaration
(its body is a definition)
```

> In C, declarations and definitions must occur at the beginning of a block.

> In C++ declarations and definitions can be placed ***anywhere*** an executable statement can appear, <u>except</u> that they must appear prior to the point at which they are first used.

> This improves the readability of the program.

> A variable lives only in the block, in which it was defined. This block is the scope of this variable.

C++'s Enhancements to C (Non OO)

```
int a=0;

for (int i=0; i < 100; i++){   // i is declared in for loop

    a++;

    int p=12;            // Declaration of p

    ...                  // Scope of p

}
```

- Variable **i** is created at the beginning of the for loop once.
- Variable **p** is created **100** times.

C++'s Enhancements to C (Non OO)

- Scope Operator **::**

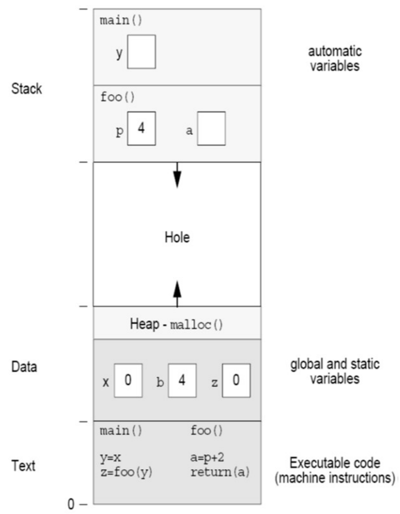
  A definition in a block can hide a definition in an enclosing block or a global name.

  It is possible to use a hidden global name by using the scope resolution operator **::**

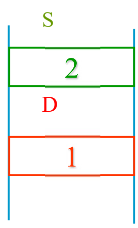
```
int y=0;           // Global y
int x=1;           // Global x
void f(){          // Function is a new block
    int x=5;       // Local  x=5, it hides global x
    ::x++;         // Global x=2
    x++;           // Local x=6
    y++;           // Global y=1
}
```

Process Memory Model

```
int x;
static int b = 4;

main() {
  int y;
  static int z;

 y = b;
  z = foo(y);
}

foo( int p ) {
  int a;
  a = p + 2;
  return(a);
}
```




```
int x=1;
void f(){
   int x=2; // Local  x
   ::x++;    // Global x is 2
}
```



S stands for *Stack*
D stands for *Data*

```
int i=1;
int main() {
  int i=2;
  {
       int n=i ;
       int i = 3 ;
       cout << i << " " << ::i << endl ;
       cout << n << "\n" ;
   }
  cout << i << " " << ::i << endl;
  return 0 ;
}
```

```
int i=1;
int main() {
  int i=2;
  {
       int n=i ;
       int i = 3 ;
       cout << i << " " << ::i << endl ;
       cout << n << "\n" ;
   }
  cout << i << " " << ::i << endl;
  return 0 ;
}
```

```
3 1
2
2 1
```

Scope Operator ::

- Like in C, in C++ the same operator may have more than one meaning.
- The scope operator has also many different tasks.

C++ AS A BETTER C

INLINE FUNCTIONS

inline functions

> In C, macros are defined by using the #define directive of the preprocessor.

> In C++ macros are defined as normal functions. Here the keyword inline is inserted before the declaration of the function.

> Remember the difference between normal functions and macros:

> A normal function is placed in a separate section of code and a call to the function generates a jump to this section of code.

> Before the jump the return address and arguments are saved in memory (usually in stack).

inline functions

> When the function has finished executing, return address and return value are taken from memory and control jumps back from the end of the function to the statement following the function call.

> The advantage of this approach is that the same code can be called (executed) from many different places in the program. This makes it unnecessary to duplicate the function's code every time it is executed.

> There is a disadvantage as well, however.

> The function call itself, and the transfer of the arguments take some time. In a program with many function calls (especially inside loops), these times can add up and decrease the performance.

inline functions

```
#define sq(x)  (x*x)

#define max(x,y)  (y<x ? x : y)
```

inline functions

```
#define sq(x)  (x*x)
inline int SQ(int x){return (x*x); }
#define max(x,y)  (y<x ? x : y)
inline int max(int x,int y){
    return (y<x ? x : y);
}
```

inline functions

```
#define sq(x)  (x*x)
inline int SQ(int x){return (x*x); }
#define max(x,y)  (y<x ? x : y)
inline int max(int x,int y){
    return (y<x ? x : y);
}
```

> An inline function is defined using almost the same syntax as an ordinary function.

> However, instead of placing the function's machine-language code in a separate location, the compiler simply inserts it into the location of the function call:

```
int j, k, l ;
. . . // Some operations over k and l
j = max( k, l ) ;
```

inline functions

```
#define sq(x)  (x*x)
inline int SQ(int x){return (x*x); }
#define max(x,y)  (y<x ? x : y)
inline int max(int x,int y){
    return (y<x ? x : y);
}
```

> An inline function is defined using almost the same syntax as an ordinary function.

> However, instead of placing the function's machine-language code in a separate location, the compiler simply inserts it into the location of the function call:

```
int j, k, l ;
. . . // Some operations over k and l
j= (k<l ? k : l) ;
```

inline functions

The decision to inline a function must be made with some care.

- If a function is more than a few lines long and is called many times, then inlining it may require much more memory than an ordinary function.
- It's appropriate to inline a function when it is short, but not otherwise. If a long or complex function is inlined, too much memory will be used and not much time will be saved.

inline functions

- Advantages
  - Debugging
  - Type checking
  - Readable

C++ AS A BETTER C

DEFAULT FUNCTION ARGUMENTS

Default Function Arguments

> A programmer can give default values to parameters of a function. In calling of the function, if the arguments are not given, default values are used.

```
int exp(int n,int k=2){
    if(k == 2)
      return (n*n) ;
    else
      return ( exp(n,k-1)*n ) ;
}
```

```
exp(i+5)
   // (i+5)* (i+5)
exp(i+5,3)
   // (i+5)^3
```

Example

> In calling a function argument must be given from left to right without skipping any parameter

```
void f(int i, int j=7) ; // right
void g(int i=3, int j) ; // wrong
void h(int i, int j=3,int k=7) ; // right
void m(int i=1, int j=2,int k=3) ; // right
void n(int i=2, int j,int k=3) ; // right ? wrong
```

Example

```
void fun(int i=1, int j=2,int k=3) ;
```

> fun() → fun(1,2,3)

> fun(2) → fun(2,2,3)

> fun(3,4) → fun(3,4,3)

> fun(5,6,7) → fun(5,6,7)

C++ AS A BETTER C

DECLARATION VS DEFINITION

Function Declarations and Definitions

> C++ uses a stricter type checking.

> In function declarations (prototypes) the data types of the parameters must be included in the parentheses.

```
// declaration
char grade (int, int, int);
int main() {
   :
}
// definition
char grade (int exam_1, int exam_2, int final_exam) {
  :          // body of function
}
```

Function Declarations and Definitions

- In C++ a return type must be specified; a missing return type does not default to int as is the case in C.
- In C++, a function that has no parameters can have an empty parameter list.

```
int print(void);   /* C style  */
int print();        // C++ style
```

Hint

- In principle all C code could be recompiled as C++ .
- In practice, C++ has somewhat ***stricter type checking*** and the compiler is able to detect ***more bugs*** and issue ***more warnings***.
- So recompiling C code using a C++ compiler is a way of ***cleaning up*** the existing code.
- The changes that have to be introduced into the source code at that stage are mostly bug fixes and stricter type enforcement.
- ***All this work is definitely worth the effort*** !

C++ AS A BETTER C

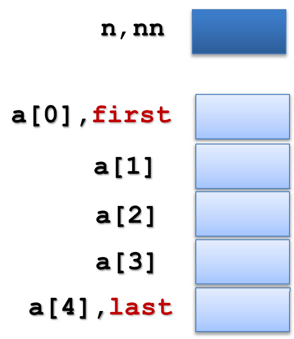
REFERENCE OPERATOR

Reference Operator : &

> This operator provides an alternative name for storage

> There are two usages of the operator

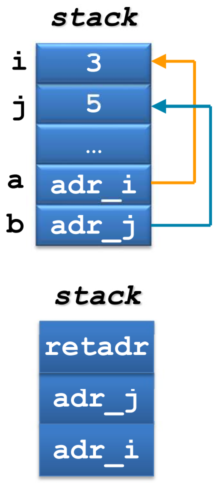
❶
```
int n ;
int& nn = n ;
double a[5] ;
double& first = a[0] ;
double& last = a[4] ;
const char& new_line = '\n';
```



❷ > Parameters Passing: Consider **swap()** function

```
void swap(int *a, int *b){
     int temp = *a ;
     *a = *b ;
     *b = temp ;
}
```

```
int main(){
  int i=3,j=5 ;
  swap(&i,&j) ;
  cout << i << " " << j << endl;
}
```



5 3

```
void swap(int& a,int& b){
     int temp = a ;
     a = b ;
     b = temp ;   }
```

```
int main(){
   int i=3,j=5 ;
   swap(i,j) ;
   cout << i << " " << j << endl ;
}
```

5 3

```
void shift(int& a1,int& a2,int& a3,int& a4){
     int tmp = a1 ;
     a1 = a2 ;
     a2 = a3 ;
     a3 = a4 ;
     a4 = tmp ;
}
```

```
int main(){
     int x=1,y=2,z=3,w=4;
     cout << x << y << z << w << endl;
     shift(x,y,z,w) ;
     cout << x << y << z << w << endl;
     return 0 ;
}
```

```
int squareByValue(int a){
   return (a*a) ;
}
```

```
void squareByReference(int& a){
    a *= a ;
}
```

```
int main(){
   int x=2, y=3, z=4 ;
   squareByPointer(&x);
   cout << x << endl;
   squareByReference(y);
   cout << y << endl;
   z = squareByValue(z);
   cout << z << endl;
}
```

```
void squareByPointer(int *aPtr){
    *aPtr = *aPtr**aPtr ;
}
```

```
4
9
16
```

`const` Reference

> To prevent the function from changing the parameter accidentally, we pass the argument as constant reference to the function.

```
struct Person{
    char name [40];
    int reg_num;
};
void print (const Person &k) {
    cout << "Name: " << k.name << endl;
    cout << "Num: " << k.reg_num << endl;
}
int main(){
    Person lost;
    strcpy(lost.name,"Jack Shephard");
    lost.reg_num=4815162342;
    print(lost);
    return 0;
}
```

Instead of 44 bytes only 4 bytes (address) are sent to the function.

Return by reference

> By default in C++, when a function returns a value:

`return` ***`expression`*`;`**

– ***`expression`*** is evaluated and its value is copied into stack.

– The calling function reads this value from stack and copies it into its variables.

> An alternative to "return by value" is "return by reference", in which the value returned is not copied into stack.

> One result of using "return by reference" is that the function which returns a parameter by reference can be used on the left side of an assignment statement.

Return by reference

```
int&  max( const int a[], int length) {
  int i=0;
  for (int j=0 ; j<length ; j++)
      if (a[j] > a[i])
          i = j;
  return a[i];
}
int main()  {
  int array[ ] = {12, -54 , 0 , 123, 63};
  max(array,5) = 0;
  :
```

`const` return parameter

> To prevent the calling function from changing the return parameter ***accidentally***, `const` qualifier can be used.

```
const int&  max( int a[ ], int length){
   int i=0;
   for (int j=0 ; j<length ; j++)
       if (a[j] > a[i]) i = j;
   return a[i];
}
```

`const` return parameter

> This function can only be on right side of an assignment

```
int main() {
 int array[] = {12, -54 , 0 , 123, 63};
 int largest;
 largest = max(array,5);
 cout << "Largest element is " << largest;
 return 0;
}
```

Never return a local variable by reference!

> Since a function that uses “return by reference” returns an actual memory address, it is important that the variable in this memory location remain in existence after the function returns.

> When a function returns, local variables go out of existence and their values are lost.

```
int& f( ) {
 int i;
 :
 return  i;
}
```

Never return a local variable by reference!

> Local variables can be returned by their values

```
int f( ) {
  int i;
  :
  return  i; // OK.
}
```

C++ AS A BETTER C

new AND delete OPERATORS

new/delete

- In ANSI C, dynamic memory allocation is normally performed with standard library functions **`malloc`** and **`free`**.
- The C++ **`new`** and **`delete`** operators enable programs to perform dynamic memory allocation more easily.

new/delete

- The most basic example of the use of these operators is given below. An int pointer variable is used to point to memory which is allocated by the operator **`new`**.
- This memory is later released by the operator **`delete`**.
- in C:

```
int *p ;
p = (int *) malloc(N*sizeof(int)) ;
free(p) ;
```

- in C++:

```
int *p ;
p = new int[N] ;
delete []p ;
```

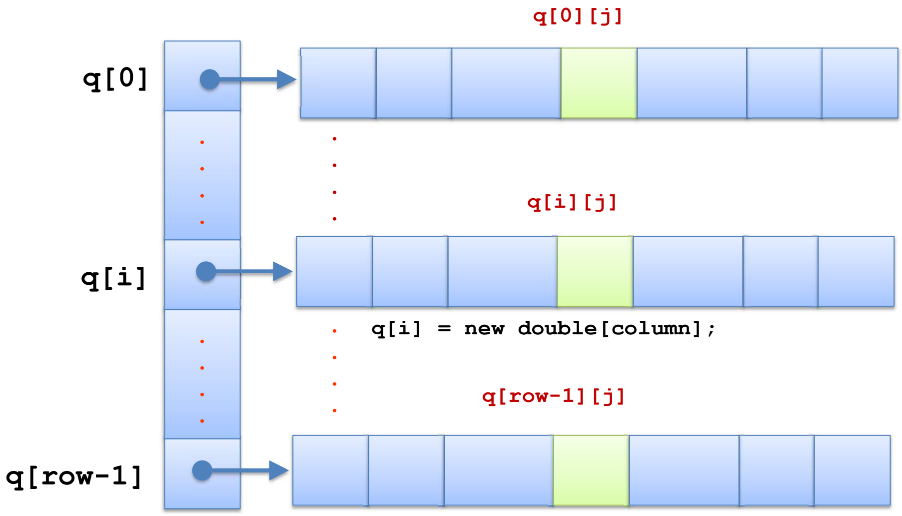
Be careful!

```
int *p,*q ;
p = new int[9]{1,2,3,0} ;
q = new int(9) ;
```

> Two Dimensional Array

```
double ** q ;
q = new double*[row] ;
// matrix size is rowxcolum
for(int i=0;i<row;i++)
   q[i] = new double[column] ;
..
for(int i=0;i<row;i++)
   delete []q[i] ;
delete []q ;
```

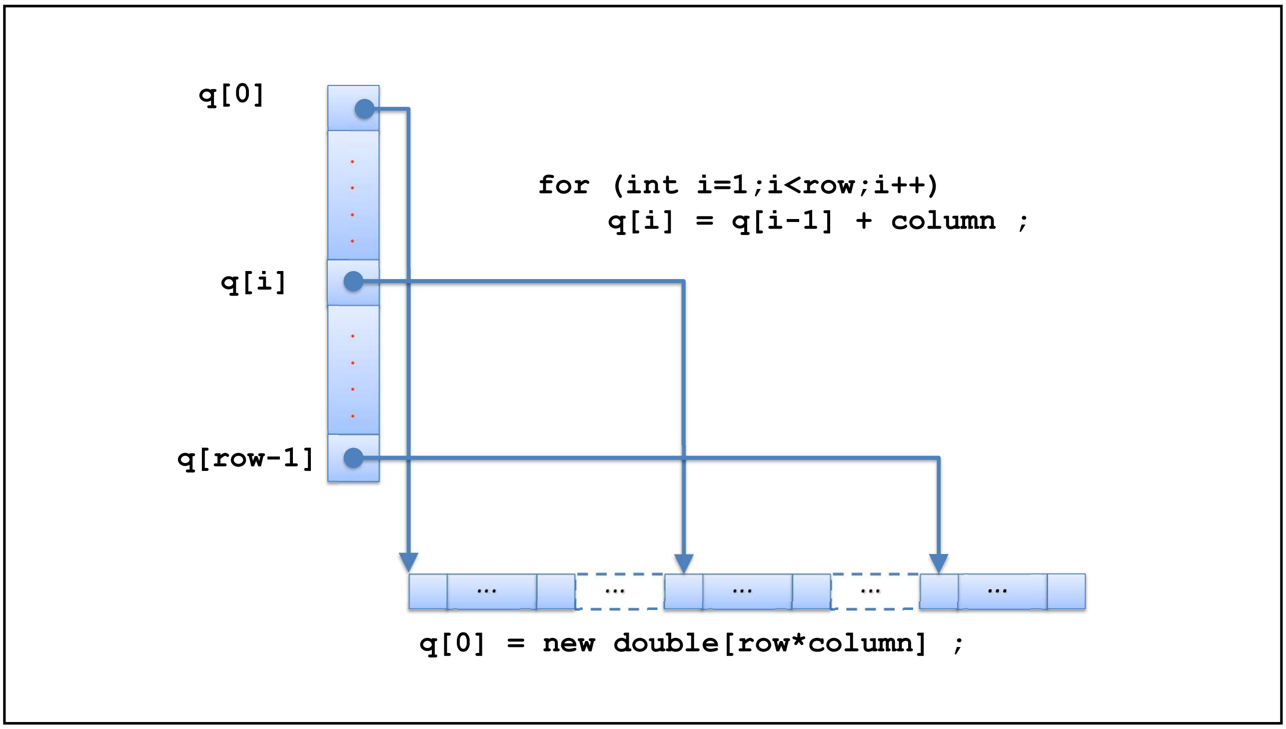
i[th] row *j*[th] column: q[*i*][*j*]



> Two Dimensional Array

❷
```
double **q;
p = new double*[row] ;
// matrix size is rowxcolumn
q[0] = new double[row*column] ;
for (int i=1;i<row;i++)
    q[i] = q[i-1] + column ;
 …..
delete []q[0] ;
delete []q ;
```

i^{th} row j^{th} column: q[i][j]




```
double ** q ;
memoryAlign= column % 4;
memoryWidth= ( memoryAlign == 0 ) ?
          column:(column+ 4-memoryAlign);
q[0] = new double[row*memoryWidth] ;
for(int i=0;i<row;i++)
     q[i] = q[i-1] + memoryWidth ;
..
delete []q[0] ;
delete []q ;
```

C++ AS A BETTER C

FUNCTION OVERLOADING

Function Overloading

> Function Overloading

```
double average(const double a[],int size) ;
double average(const int a[],int size) ;
double average(const int a[],
               const double b[],int size) ;

double average(const int a[],int size) {
  double sum = 0.0 ;
  for(int i=0;i<size;i++)
      sum += a[i] ;
  return ((double)sum/size) ;
}
```

```
double average(const double a[],int size) {
   double sum = 0.0 ;
   for (int i=0;i<size;i++)
       sum += a[i] ;
   return (sum/size) ;
}
double average(const int a[],
               const double b[],int size) {
   double sum = 0.0 ;
   for (int i=0;i<size;i++)
       sum += a[i] + b[i] ;
   return (sum/size) ;
}
```

```
int main() {
  int w[5]={1,2,3,4,5} ;
  double x[5]={1.1,2.2,3.3,4.4,5.5} ;
❶ cout << average(w,5);
❷ cout << average(x,5);
❸ cout << average(w,x,5);
  return 0 ;
}
```

C++ AS A BETTER C

FUNCTION TEMPLATES

Function Templates

> Function Templates

```
template <typename T>
void printArray(const T *a,const int size){
     for (int i=0;i < size;i++)
          cout << a[i] << " " ;
     cout << endl ;
}
```

Function Templates

> Function Templates

```
template <typename T1,class T2>
void printArray(const T1 *a,const T2 size){
     for (int i=0;i < size;i++)
          cout << a[i] << " " ;
     cout << endl ;
}
```

```
int main() {
   int a[3]={1,2,3} ;
   double b[5]={1.1,2.2,3.3,4.4,5.5} ;
   char c[7]={'a','b','c','d','e','f','g'};
❶  printArray(a,3);
❷  printArray(b,5);
❸  printArray(c,7);
   return 0 ;
}
```

```
void printArray(int *a,const int size){
     for(int i=0;i < size;i++)
        cout << a[i] << ",″ ;
     cout << endl ;
}
void printArray(double *a,const int size){
     for(int i=0;i < size;i++)
        cout << a[i] ;
     cout << endl ;
}
void printArray(char *a,const int size){
     for(int i=0;i < size;i++)
        cout << a[i] ;
     cout << endl ;
}
```

```
template <class T>
inline T const& max (T const& a,T const& b){
   return a<b?b:a;
}
// OK. Both parameters are integer.
max(4,7)
// Error! Parameters are of different type
max(4,4.2)
```

> How to correct
> ❶ `max<double>(4,4.2) // OK!`
> ❷ `max(static_cast<double>(4),4.2) // OK!`

Nontype Template Parameters

```
template <typename T, int VAL>
T add_val(T x){
     return x + VAL ;
}
template <typename T>
T add_val(T x, int VAL){
     return x + VAL ;
}
double y=42.0;
double z = add_val<double,108>(y);
double w = add_val<double,42>(y);
double m= add_val(y,4);
```

Nontype Template Parameters

- In addition to type parameters, it is also possible to use ***nontype*** parameters.
- A ***nontype*** parameter is then considered as part of type.
- For example, for the standard class **`bitset<>`** you can pass the number of bits as the template argument.
- The following statements define two bitfields, one with 32 bits and one with 50 bits:
  - `bitset<32> flags32; //bitset with 32-bits`
  - `bitset<50> flags50; //bitset with 50-bits`
- These bitsets have different types because they use different template arguments.
- Thus, you can't assign or compare them (except if a corresponding type conversion is provided).

Default Template Parameters

> Templates classes may have default arguments

```
template <class T,class container=vector<T> >
class MyClass{
    ...
};
```

> If you pass only one argument, the default parameter is used as second argument:

```
MyClass<int> x1;
// equivalent to MyClass<int,vector<int>>
```

> Note that default template arguments may be defined in terms of previous arguments.

Default Template Parameters

```
template <class T=int,class container=vector<T>>
class MyClass{
    ...
};
```

Explicit Initialization for Template Types

```
int i1; // undefined value
int i2 = int(); // initialized with zero
```

> This feature is provided to enable you to write template code that ensures that values of any type have a certain default value.

```
template <class T>
void f() {
  T x = T();
  ...
}
```

C++ AS A BETTER C

OPERATOR OVERLOADING

Operator Overloading

> In C++ it is also possible to overload the built-in C++ operators such as **+**, **−**, **=** and **++** so that they too invoke different functions, depending on their operands.

> The operator **+** in **a+b** will add the variables if **a** and **b** are integers, but will call a different function if **a** and **b** are variables of a user defined type.

Operator Overloading: Rules

> You can't overload operators that don't already exist in C++

> You can not change numbers of operands.

– A binary operator (for example +) must always take two operands.

> You can not change the precedence of the operators.

***** comes always before **+**

> Everything you can do with an overloaded operator you can also do with a function.

– However, by making your listing more intuitive, overloaded operators make your programs easier to write, read, and maintain.

> Operator overloading is mostly used with objects. We will discuss this topic later more in detail.

Operator Overloading

> Functions of operators have the name operator and the symbol of the operator. For example the function for the operator + will have the name **`operator+`**:

```
struct SComplex {
 float real,img;
};
SComplex operator+(const SComplex& lhs,
                   const SComplex& rhs){
  SComplex result;
  result.real= lhs.real + rhs.real;
  result.img =  lhs.img + rhs.img;
  return result;
}
```

```
int main(){
  SComplex c1={1,2},c2 ={5,1};
  SComplex c3;
  c3=c1+c2;   // +(c1,c2)
}
```

C++ AS A BETTER C

NAMESPACE

namespace

- When a program reaches a certain size it's typically broken up into pieces, each of which is built and maintained by a different person or group.
- Since C effectively has a single arena where all the identifier and function names live, this means that all the developers must be careful not to accidentally use the same names in situations where they can conflict.
- The same problem come out if a programmer try to use the same names as the names of library functions.
- Standard C++ has a mechanism to prevent this collision: the namespace keyword. Each set of C++ definitions in a library or program is "wrapped" in a namespace, and if some other definition has an identical name, but is in a different namespace, then there is no collision.

namespace

```
namespace programmer1{
      int iflag;
      void g(int);
      :
}
namespace programmer2{
      int iflag;
      :
}
```

Accessing Variables

```
programmer1::iflag = 3;
programmer2::iflag = -345;
programmer1::g(6);
```

- If a variable or function does not belong to any namespace, then it is defined in the global namespace.
- It can be accessed without a namespace name and scope operator.

Accessing Variables

- This declaration makes it easier to access variables and functions, which are defined in a namespace.

```
using programmer1::iflag;
// programmer1::iflag=3;
iflag = 3;
programmer2::iflag = -345;
programmer1::g(6);
```

Accessing Variables

```
// applies to all elements in the namespace
using namespace programmer1;
// programmer1::iflag=3;
iflag = 3;
// programmer1's function g
g(6);
programmer2::iflag =  -345;
```

namespace

```
#include <iostream>
namespace F {
  float x = 9;
}
namespace G {
  using namespace F;
  float y = 2.0;
  namespace INNER_G {
    float z = 10.01;
  }
}
```

```
int main() {
 float x = 19.1;
 using namespace G;
 using namespace G::INNER_G;
 cout << "x = "
      << x << std::endl;
 cout << "y = "
      << y << std::endl;
 cout << "z = "
      << z << std::endl;
 return 0;
}
```

mod02/namespace1.cpp

namespace

```cpp
#include <iostream>
namespace F {
  float x = 9;
}
namespace G {
  using namespace F;
  float y = 2.0;
  namespace INNER_G {
    long x = 5L;
    float z = 10.01;
  }
}
```

```cpp
int main() {
 using namespace G;
 using namespace G::INNER_G;
 cout << "x = "
      << x << std::endl;
 cout << "y = "
      << y << std::endl;
 cout << "z = "
      << z << std::endl;
 return 0;
}
```

mod02/namespace2.cpp

namespace

```cpp
#include <iostream>
namespace F {
  float x = 9;
}
namespace G {
  using namespace F;
  float y = 2.0;
  namespace INNER_G {
    long x = 5L;
    float z = 10.01;
  }
}
```

```cpp
int main() {
 using namespace G;
 cout << "x = "
      << x << std::endl;
 cout << "y = "
      << y << std::endl;
 return 0;
}
```

mod02/namespace3.cpp

namespace

```
#include <iostream>
namespace F {
  float x = 9;
}
namespace G {
  float y = 2.0;
  namespace INNER_G {
    long x = 5L;
    float z = 10.01;
  }
}
```

```
int main() {
using namespace G;
cout << "x = "
      << x << std::endl;
 cout << "y = "
      << y << std::endl;
return 0;
}
```

mod02/namespace4.cpp

namespace

```
#include <iostream>
namespace F {
  float x = 9;
}
namespace G {
  float y = 2.0;
  namespace INNER_G {
    long x = 5L;
    float z = 10.01;
  }
}
```

```
int main() {
 using namespace G::INNER_G;
 cout << "x = "
      << x << std::endl;
 cout << "y = "
      << y << std::endl;
 return 0;
}
```

mod02/namespace5.cpp

namespace

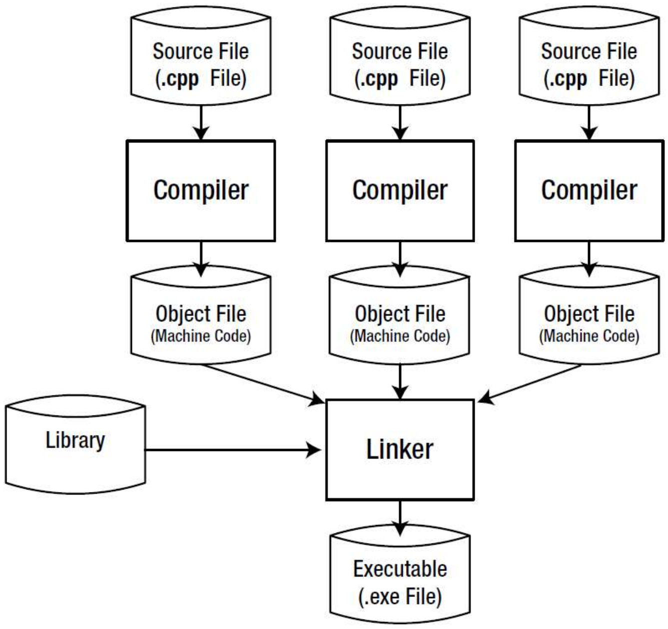
```
#include <iostream>
namespace F {
  float x = 9;
}
namespace G {
  float y = 2.0;
}
namespace G::INNER_G {
    long x = 5L;
    float z = 10.01;
}
```

```
int main() {
 using namespace
G::INNER_G;
 cout << "x = "
      << x << std::endl;
 cout << "y = "
      << y << std::endl;
 return 0;
}
```

C++ AS A BETTER C

STANDARD C++ HEADER FILES

The compile and link process



Standard C++ Header Files

> In the first versions of C++, mostly '.h' is used as extension for the header files.

> As C++ evolved, different compiler vendors chose different extensions for file names (.hpp, .H , etc.). In addition, various operating systems have different restrictions on file names, in particular on name length. These issues caused source code portability problems.

> To solve these problems, the standard uses a format that allows file names longer than eight characters and eliminates the extension.

> For example, instead of the old style of including **iostream.h**, which looks like this: **`#include <iostream.h>`**, you can now write: #include <iostream>

Standard C++ Header Files

- The libraries that have been inherited from C are still available with the traditional '.h' extension. However, you can also use them with the more modern C++ include style by puting a "c" before the name. Thus:

  `#include <stdio.h>` becomes **`#include <cstdio>`**

  `#include <stdlib.h>` becomes **`#include <cstdlib>`**
- In standard C++ headers all declarations and definitions take place in a namespace : **`std`**
- Today most of C++ compilers support old libraries and header files too.
- So you can also use the old header files with the extension '.h'.
- For a high-quality program prefer always the new libraries.

I/O

- Instead of library functions (printf, scanf), in C++ library objects are used for IO operations.
- When a C++ program includes the **`iostream`** header, four objects are created and initialized:
  - **`cin`**

    handles input from the standard input, the keyboard.
  - **`cout`**

    handles output to the standard output, the screen.
  - **`cerr`**

    handles unbuffered output to the standard error device, the screen.
  - **`clog`**

    handles buffered error messages to the standard error device

Using **cout** Object

- To print a value to the screen, write the word **cout**, followed by the insertion operator (**<<**).

```
// Header file for the cout object
#include<iostream>
int main() {
   int i=5;
   float f=4.6;
   std::cout << "Integer Number = " << i
             << " Real Number= " << f;
   return 0;
}
```

Using **cin** Object

- The predefined **cin** stream object is used to read data from the standard input device, usually the keyboard.
- The **cin** stream uses the **>>** operator, usually called the "get from" operator.

```
#include<iostream>
using namespace std;
int main() {
   int i,j;
   cout << "Give two numbers \n";
   cin >> i >> j;
   cout << "Sum= " << i + j << "\n";
   return 0;
}
```

`std` namespace

```
#include <string>
#include <iostream>
using namespace std;
int main() {
 string test;
 while(test.empty() || test.size() <= 5){
     cout << "Type a string longer string. "
          << endl;
     cin >> test;
}
```

```
printf("%s",test.c_str())
```

`bool` Type

> The type **`bool`** represents boolean (logical) values: true and false

> Before **`bool`** became part of Standard C++, everyone tended to use different techniques in order to produce Boolean-like behavior.

> These produced portability problems and could introduce subtle errors.

> Because there's a lot of existing code that uses an **`int`** to represent a flag, the compiler will implicitly convert from an **`int`** to a **`bool`** (non-zero values will produce **`true`** while zero values produce **`false`**).

> Do not prefer to use integers to produce logical values.

`bool` Type

```
// Boolean variable: is_greater
bool is_greater;
// Assigning a logical value
is_greater = false;
int a,b;
...
// Logical operation
is_greater = a > b;
// Conditional operation
if (is_greater)
    ...
```

C++ AS A BETTER C

const

constant

- In standard C, preprocessor directive **`#define`** is used to create constants:
  `#define PI 3.14`
- C++ introduces the concept of a named constant that is just like a variable, except that its value cannot be changed.
- The modifier **`const`** tells the compiler that a name represents a constant:

```
const int MAX = 100;
…
MAX = 5; // Compiler Error!
```

- **`const`** can take place before (left) and after (right) the type. They are always (both) allowed and equivalent.

```
int const MAX = 100;
// The same as const int MAX = 100;
```

constant

- Decreases error possibilities.
- To make your programs more readable, use uppercase font for constant identifiers.

Use of constant – 1

> Another usage of the keyword const is seen in the declaration of pointers. There are ***three*** different cases:

a) The data pointed by the pointer is constant, but the pointer itself however may be changed.

```
const char *p = "ABC";
```

– **p** is a pointer variable, which points to chars.

– The **const** word may also be written after the type:

```
char const *p = "ABC";
```

> Whatever is pointed to by **p** may not be changed: the chars are declared as **const**.

> The pointer **p** itself however may be changed.

```
*p = 'Z';    // Compiler Error!
p++;    //OK
```

Use of constant – 2

b) The pointer itself is a const pointer which may not be changed. Whatever data is pointed to by the pointer may be changed.

```
// Pointer is constant, data may change
char * const sp = "ABC";
// OK, data is not constant
*sp = 'Z';
// Compiler Error!
// Because pointer is constant
sp++;
```

Use of constant – 3

c) Neither the pointer nor what it points to may be changed

The same pointer definition may also be written as follows:

```
char const * const ssp = "ABC";
const char * const ssp = "ABC";
// Compiler Error! Because data is constant
*ssp = 'Z';
// Compiler Error! Because pointer is const
ssp++;
```

> The definition or declaration in which const is used should be read from the variable or function identifier back to the type identifier: "**ssp** is ***a const pointer*** to ***const*** characters"

C++ AS A BETTER C

CAST OPERATORS

Casts

- Traditionally, C offers the following cast construction:

  `(typename) expression`

  Example: `f = (float)i / 2;`
- C++ initially also supported the function call style cast notation:

  `typename(expression)`

  Example: Converting an integer value to a floating point value

```
int i=5;
float f;
f = float(i)/2;
```

- But, these casts are now called ***old-style casts***, and they are deprecated. Instead, ***four new-style casts*** were introduced.

Casts: `static_cast`

- The `static_cast<type>(expression)` operator is used to convert one type to an acceptable other type.

```
int i=5;
float f;
f= static_cast<float>(i)/2;
```

Casts: const_cast

> The **const_cast<type>(expression)** operator is used to do away with the const-ness of a (pointer) type.

> In the following example **p** is a pointer to constant data, and **q** is a pointer to non-constant data. So the assignment **q = p** is not allowed.

```
const char *p = "ABC";   // p points to constant data
char *q;     // data pointed by q may change
q = p;       // Compiler Error! Constant data may change
```

> If the programmer wants to do this assignment on purpose then you must use the **const_cast<>** operator:

```
q = const_cast<char *>(p);
*q = 'X';     // Dangerous?
```

Casts: reinterpret_cast

> The **reinterpret_cast<type>(expression)** operator is used to reinterpret byte patterns. For example, the individual bytes making up a structure can easily be reached using a **reinterpret_cast**

```
struct S {
   int i1,i2;
};
```

```
int main() {
 S x;
 x.i1=1;
 x.i2=2;
 unsigned char *xp;
 xp = reinterpret_cast<unsigned char *> (&x);
 for (int j=0; j<8; j++)
     std::cout << static_cast<int>(*xp++);
 return 0;
}
```

Casts: `dynamic_cast`

- The **`dynamic_cast<>()`** operator is used in the context of inheritance and polymorphism. We will see these concepts later. The discussion of this cast is postponed until the section about polymorphism.
- Using the cast-operators is a dangerous habit, as it suppresses the normal type-checking mechanism of the compiler.
- It is suggested to prevent casts if at all possible.
- If circumstances arise in which casts have to be used, document the reasons for their use well in your code, to make double sure that the cast is not the underlying cause for a program to misbehave.

C++ AS A BETTER C

INTEGER TYPES

Integer Types

> C++11 standardized a fifth integer type: **long long**

– guaranteed to be at least 64-bits large

> Fixed-sized integer types were added in C++11.

> These types belong to the std namespace and can be included through the cstdint standard library header.

```
#include <cstdint>
using namespace std;
int8_t myInt8 = 0; // 8 bits
int16_t myInt16 = 0; // 16 bits
int32_t myInt32 = 0; // 32 bits
int64_t myInt64 = 0; // 64 bits
```

Integer Types

Type Name	Memory Used	Size Range
int8_t	1 byte	-128 to 127
uint8_t	1 byte	0 to 255
int16_t	2 bytes	-32,768 to 32,767
uint16_t	2 bytes	0 to 65,535
int32_t	4 bytes	-2,147,483,648 to 2,147,483,647
uint32_t	4 bytes	0 to 4,294,967,295
int64_t	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
uint64_t	8 bytes	0 to 18,446,744,073,709,551,615